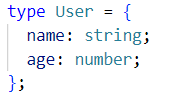
📘 Chapter 7: Type vs Interface

Both type and interface allow you to define the shape of an object in TypeScript.

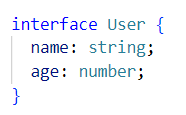
They might look the same, but they function very differently beneath the surface.

**Basic Syntax:**

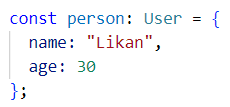
**Type -**



**Interface -**



**Usage in object -**



***Similarities –***

✅ Both can describe object shapes  
✅ Both support optional properties  
✅ Both support readonly properties  
✅ Both support function types  
✅ Both support extending other types/interfaces

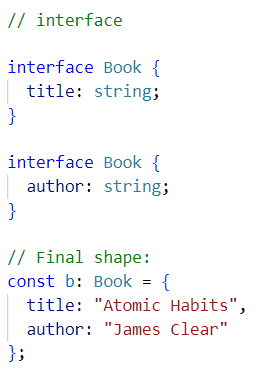
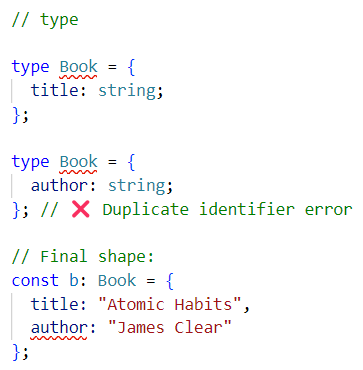
***Differences (with examples) -***

**1. Re-opening (Declaration Merging) -**

interface allows you to define it in multiple places — and it merges automatically

type cannot be re-opened once declared

Example:

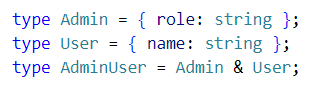


**2. Union & Intersection -**

type supports unions.

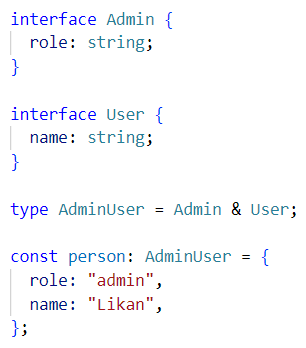


type supports intersections.



Interface Supports Intersections.

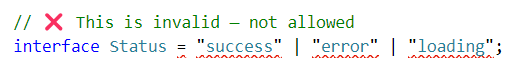
You can intersect interfaces just like types using the & operator.



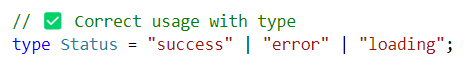
Even though Admin and User are defined using interface, the intersection (Admin & User) is defined using type, which is totally valid.

### interface does **not** support union types directly.

You **cannot** define a union like this using interface:



That’s **only allowed with** type:



### Why?

* interface is designed to define the **structure of an object**.
* type is more flexible and can describe **primitives, unions, intersections, functions, tuples, etc.**

## **3. Computed Properties + Conditional Types** **(Advanced Meta-Programming)**

### Only type (not interface) can do this kind of advanced logic.

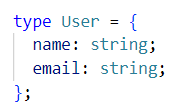
### Part 1: **Computed Properties with Mapped Types**

### 

### What’s happening here?

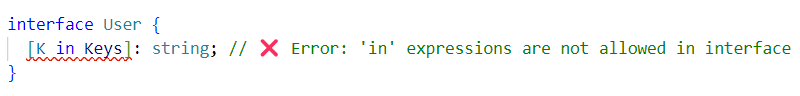
* Keys is a union type: "name" | "email"
* [K in Keys] means: "for each K in Keys"

User becomes -



This is **meta-programming** - generating types **dynamically** using logic.

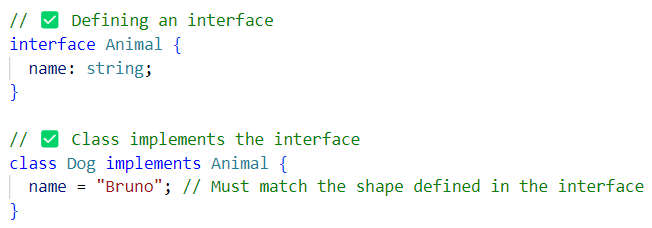
### 🚫 interface can't do this -



## **4. Implementing & Extending in Classes**

### Both interface and type can be used with class via implements.

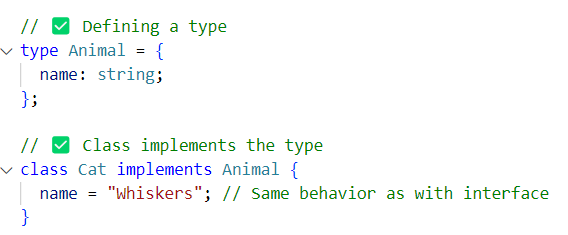
### Example 1: Using interface with class



**🧠 What happens here?**

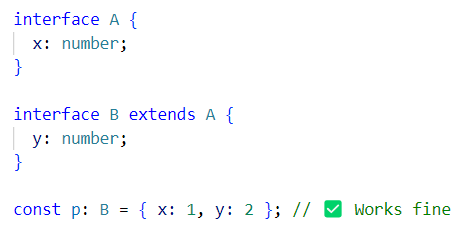
* The class Dog is **guaranteed** to have a name property of type string.
* TypeScript will show an error if name is missing or the type is wrong.

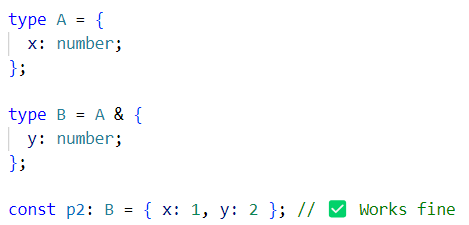
### Example 2: Using type with class



🧠 You get **the same result** as using interface.

Example 3 -





Same result — just different approach.

### 🛠 ****Use**** interface ****when -****

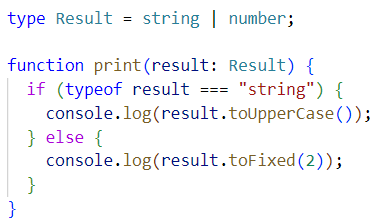
✅ You're defining **object shapes that may grow** (especially in large or scalable apps)  
✅ You need **declaration merging** (like extending or reopening the same interface)  
✅ You're building **React component props** or **class blueprints.** **Because** interface **is made for object shapes** — and props in React **are just objects**. interface lets you define the shape of that object in a clean, extendable way. When you define a class, interface works well to describe what structure the class must follow especially when you want to use implements.  
✅ You want others to **extend** your structure easily via extends

### ⚡ ****Use**** type ****when -****

✅ You want **unions** (A | B) or **intersections** (A & B)  
✅ You're building **utility types**, like extracting or transforming types  
✅ You need **conditional types** (e.g. T extends U ? X : Y)  
✅ You want a **more expressive, strict, or computed structure**

Regular Union Vs Discriminated Union -

### 1. Regular Union (❌ No discriminant key)

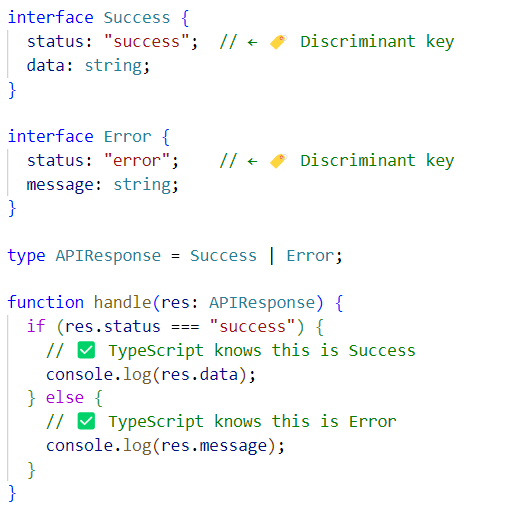


Here you rely on typeof or **manual checks** to narrow the type.

### 2. Discriminated Union or 🏷️ Tagged Union

### What is a **Tagged Union** (Discriminated Union)?

A **Tagged Union** is a **union of types** where **each type has a unique "tag" property** (usually a string literal like "type", "kind", or "status"). This tag helps TypeScript **automatically figure out** which specific type you're working with during **type narrowing**.

  
In this example status is the **discriminant key** (aka the **tag**). APIResponse is the union type.

This is called a **Discriminated Union** because each type in the union is "tagged" with a unique status key. Based on the value of status, we can **narrow down the type**.

**Type narrowing** helps TypeScript understand the **exact or specific type at compile time,** enabling safer and more accurate code

A discriminated union:

* Requires a common literal field (status, type, etc.).
* Allows type-safe checks without manual type assertions.
* Keeps code clean, safe, and readable.
* Works best when combining multiple object shapes.

Interview Questions

***Q1: Can you use a type and interface interchangeably?***

Sometimes yes (for object shapes), but only type supports unions/conditions.

***Q2: Can interfaces merge? What about types?***

Only interfaces merge automatically.

***Q3: When would you choose type over interface?***

When using advanced features like unions, mapped types, or conditional types.

***Q4: Which one is preferred in public libraries (like React)?***

interface — because it's easier to extend via declaration merging.